A Reliable Instant Messenger in Erlang: Design and Evaluation

Mario Moro Hernández, Natalia Chechina, and Phil Trinder

School of Computing Science
The University of Glasgow
Glasgow G12 8QQ
UK

Technical Report TR-2015-002

December 17, 2015

Contents

Chapter 1

Introduction

This document describes the design and evaluation of two Erlang-based instant messenger systems using Distributed Erlang (D-Erlang) and Scalable Distributed Erlang (SD-Erlang). The purpose of these systems is to serve as real-world benchmarks to test the performance of the SD Erlang library.

The Erlang programming language was invented by Joe Armstrong, Robert Virding, and Mike Williams at Ericsson Computer Science Lab in 1986. It was conceived '*as a tool to get the job done*' [Arm07a], being this job to program the second generation of the Ericsson AXE switches. At the beginning it was a logic programming language, derived from Prolog [AVW92], designed for '*writting programs that "run forever"*' [Arm07a], and with the aim of simplifying the programming of telephony applications. Functional language features such as list comprehensions or higher-order functions were added few years later. Similarly, support for distribution was added in 1993, years after the first release, and used in production for the first time in 1995 with the AXD301 switch [Arm07b].

Today, Erlang is a soft real-time functional language that offers support for concurrency, distribution and fault tolerance. It also allows the injection of code updates without having to stop the application.

Scalable Distributed Erlang (SD Erlang) is an extension of the Erlang distribution model that let push this limit further, allowing the systems to either scale more, or given the same size, to use less resources than their distributed Erlang counterparts [CLTG14]. Given the context described above, SD Erlang is particularly useful to increase the scalability of the IM architectures. SD-Erlang has been developed in the RELEASE project, within the 7th Framework Programme of the European Commission. The different tasks performed included the execution of a battery of benchmarks to validate this SD Erlang technology. The results obtained to date show that applications written in SD Erlang scale better than their D-Erlang equivalents, penalising neither their performance, nor their reliability models [CLG+15].

Glossary

Architecture Conceptual model that defines the structure, behaviour and more views of a system.

Distributed Erlang System (Also D-Erlang System). System formed out of a number of Erlang nodes that communicate with each other. The system uses standard features of the Erlang language to write distributed programs.

Scalable Distributed Erlang System (Also SD-Erlang System). System formed out of a number of Erlang nodes that communicate with each other. The system uses features of the Scalable Distributed Erlang library to write distributed programs.

***Process*_Supervisor** Process that monitors all the processes inside the node which hosts it, and supervisor processes at children nodes (if any).

Chapter 2

Instant Messenger Design and Implementation

2.1 Idiosyncrasy of the Erlang/OTP Applications

The concept of Erlang significantly differes from other programming languages like C or Java. Similarly to other functional languages such as Haskell the computations are carried out by functions. However, there is a particular type of functions that remain continuously calling to themselves after performing some computation. These are called Erlang processes (or simply processes).

An application in Erlang is basically a collection of processes that interact with each other to implement a set of requirements. The result of these interactions is, obviously, a certain output. But since Erlang was designed for fault-tolerance, true applications in Erlang are also implement a supervision tree [CT09]. That is, Erlang applications divide the requirements between different processes that are deployed as a hierarchical implementation of the *supervisor/worker* design pattern.

As it happens for any other design-patterns, the workers and supervisors have very defined structures that usually are common within applications. For this reason the OTP framework supplies standardised templates for these tasks called behaviours, that are aimed to help in the implementation of servers, supervisors, finite state machines and event handlers [CVss].

Behaviours are interesting because they facilitate the reusability of the code, they uniform the code to a standard and, as they encapsulate some features of the language, allow developers to focus on the application without having to worry too much for aspects like fault-tolerance or code hot-swapping. For these reasons industry-standard applications are implemented using behaviours.

The applications developed for this project do not use behaviours. At the design stage it was deemed that they would introduce unnecessary com-

plexity. Instead, the supervision tree is directly embedded in the server logic of the different processes that perform monitoring tasks.

The rest of the chapter describes the implementation of the instant messenger applications made for the project, the supervision tree, the deployment sequence, and two auxiliary modules that, although they do not belong to the applications, have been used during the experimental sessions.

2.2 Architecture of the RD-IM and RSD-IM

At a first glance to the entities involved, there are no differences between RD-IM and RSD-IM applications. This is true to some extent. It is necessary to look the code inside the processes to perceive how both differ. The dissimilarities also affect the way these entities are organised and connected to each other. However, this policy of applying only the necessary changes, trying to keep the applications the closest possible to each other, ensures that only the differences between technologies and not the applications are tested.

The entities of the IM applications are realised in Erlang nodes (Erlang virtual machines). These nodes host the processes that perform the business model of the applications.

From the point of view of the architecture, the IM applications follow a simple client-server architecture. Again, since these are not industry-standard applications the decision was to keep the things simple, providing a server layer to support all the IM functions and a client layer to sustain the traffic generation logic.

2.2.1 Server-Side Nodes

Router. This node is in charge of connecting the client processes to the servers, so that the chat sessions can happen. In a sense, it is just an interface between clients and servers. The main type of processes found at router node are:

- A *router_process* forwards the client requests to the different servers deployed in the system. When a client logs in, the router finds the server node to which the client is going to be assigned, and then forwards the client request to the server_supervisor, which will spawn a *client_monitor* in that server node. The *router_process* is also responsible for supervising the *server_supervisor* process[1].
- On the other hand, the mission of the *router_supervisor* process is to monitor the router processes. When one *router_process* terminates unexpectedly, then the *router_supervisor* restarts it.

[1]Note that the number of router processes must be less than the number of server nodes.

- Finally, there is a *router_supervisor_monitor* process that monitors the *router_supervisor* process. These processes actually monitor each other in a circular way.

Server. The main purpose of the server node is to connect two (or more clients), thus they can interchange plain-text messages. It hosts three types of processes for this purpose:

- The *server_supervisor* is the process that ensures the reliability of the server node restoring any failed process —except for itself—, and thus the state of the node to the same state as it was before the failure of the process.

  This process also receives and handles the log-in requests, spawning a corresponding *client_monitor* process. When a client sends a chat session request, it spawns the *chat_session* process that enables the communication between the clients.

- *chat_session* process enables the communication between clients. It forwards the message sent by the sender to the intended receiver. It also sends a confirmation to the sender of the delivery of the message.

  Other responsibilities include register the chat session in the *Chat_Sessions_DB*, when two or more clients start a chat session, and unregister the chat session from the *Chat_Sessions_DB* when the chat session finishes.

- *client_monitor* keeps track of a client while this is logged in to the system. Hence, its responsibilities are: (i) to register the client in the *Client_DB* when it logs in, (ii) unregister the client from the *Client_DB* when it logs out, and (iii) notify the client that it is already logged in, if this client tries to register in the system while being registered already.

Server nodes also contains two distributed databases to store the information about the clients logged in the system and the running chat sessions (namely, *Clients_DB* and *Chat_Sessions_DB*).

The implementation of these data containers is made through two processes that own what is called *Erlang Term Storage (ETS) tables*, and they simply act as interfaces between the server processes and the and ETS tables. This provides protection and integrity to data since any request must be done sending a message with a prefixed format to the corresponding *DB* process. Other messages are simply ignored.

The reasons to use ETS tables instead of, for example, lists of records are mostly two: they are garbage-collected (providing a more efficient use of the memory of the host machines) and they are implemented as either hash tables (providing O(1) access times to items).

However, ETS tables are linked to the process that created them and for this reason they are volatile: when the owner process ceases to exist all the data contained in the ETS tables disappear. Having into account that the server processes rely heavily on the information stored in the databases, some replicas of these data structures are created and distributed amongst neighbouring server nodes.

2.2.2 Client-Side Nodes

Client. The client node hosts the client processes. These can be of two types: normal and '*doped*' client.

- Normal *client* is a simply command-line client that send messages and prints the received messages to the standard output of the terminal.
- *'*doped*' client* is a process based on the normal client, that embodies the traffic generation logic used to stress the IM architectures. The way this type of client works is fairly simple: the client receives a message that triggers a conversation with another '*doped*' client logged in the system. Then it generates a random string and sends the message to the receiver client after a random time ranging between 1 and 20 seconds. This simulates the time spent at typing a message.

  When a '*doped*' client receives a message, it generates a reply message (again a random string), simulates the typing time, and sends the message to the client that sent the first message.

  This is done for a random number of interactions that are specified at the beginning of the conversation. When the conversation is finished, the '*doped*' client notifies this event to the corresponding *chat_session* process, that terminates as if two normal clients had finished their chat session.

Client nodes contain a third type of process, called *routers_DB*, that is a container for the pids of the deployed router processes. From the perspective of the business logic, this is an auxiliary process. However, it is fundamental for RD-IM and RSD-IM clients to work, as it provides the client processes the pid of the routers. This way, clients can communicate to the server layer of the application.

Traffic Generator. Initially, these nodes are not essential to the IM applications. They were added after experiencing some performance problems with the first generation of *traffic_generator* processes.

There were several causes for these problems. These traffic generators *mark 1* were responsible for the message generation/waiting time/sending logic. Partly because of this, partly because they only kept a record of the

sender and receiver clients, they only could manage one conversation at a time. This implied that it was necessary to spawn a massive number of these processes to generate enough traffic. In addition, they had to be spawned inside the client nodes. And because they were not coded for efficiency, they were too greedy resources-wise.

The result of all these circumstances, was that client nodes ended up running out of memory and the whole traffic generation stopped to work when the traffic loads were intense.

The solution was then to move the message generation to the clients, and move lighter traffic generator processes to a different node. Now, these traffic generator processes *mark 2* are acting as conversations triggers. Each of them control up to ten conversations simultaneously, and thanks to moving this traffic generation logic to an external node, they can generate conversations between clients that belong to two different client nodes. This was something that the first generation was unable to do.

2.2.3 RSD-IM specific processes

Sections 2.2.1 and 2.2.2 describe the nodes found at both RD-IM and RSD-IM, as well as the processes that implement the business logic of the applications at these nodes. So far, this is enough to have the DErlang version of the instant messenger running. In turn, the SDErlang version could work, but it would fail performing some operations on the *client_DB* and *chat_DB* processes.

In the case of the RD-IM, these processes are accessed using globally registered names to find their pid, e.g.:

```
global:whereis_name(process_name) ! {message}
```

Since SD-Erlang modifies these global operations, the equivalent to the previous statement

```
s_group:whereis_name(s_group_name, process_name) ! {message}
```

would fail if *process_name* refers to some process registered at a different *s_group* than the group to which the caller function belongs to. This is particularly problematic when is necessary to access the *DB_replica* processes (e.g., to recover a *DB_main* process that went down).

The solution was to create a *relay* process that, essentially, multicast the message to all the *router_processes* present, since these processes have access to the registered names at *server*_N_*group* s_groups. Hence, when a certain *router_process* finds the target process, it simply forwards the message. If the target name is not accessible to the *router_process*, the message is discarded.

2.3 Supervision Tree

As it was mentioned at the beginning of section 2.2, RD-IM and RSD-IM are designed to be as closest as possible to each other. Following the general design principles for Erlang (cf. section 2.1) the two applications have their supervisor trees, that are the same in both cases. At server-side the hierarchy is as follows:

- *Router supervisor monitor* supervises *router supervisor*.
- *Router supervisor* supervises *Router supervisor monitor* and *Router process(es)*.
- *Router process* supervises *Server Supervisor(s)*.
- *Server Supervisor* supervises:
  - *Client Databases*.
  - *Chat Session Databases*.
  - *Chat session processes*.
  - *Client monitors*.

Client-side, the supervision tree is only one level deep: *Client monitor* supervises *Client processes*. Figure 2.1 shows this supervision tree graphically:

2.4 Deployment and architectural layout

The deployment sequence of the IM applications is is managed by the *launcher* module. This is nothing more than a launching script. In the

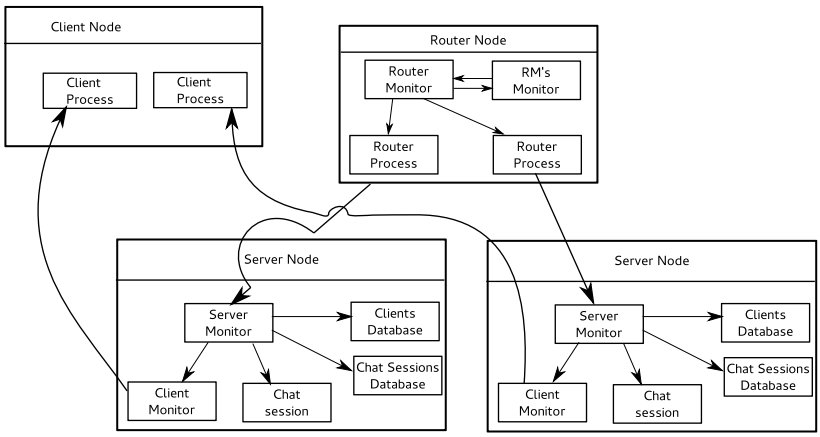

Figure 2.1: Supervision hierarchy. The arrows indicate supervision.

case of the RSD-IM, the creation of the s_groups and the assignment of the nodes to these is done at deployment stage.

Thus, the *launcher* module not only deploys the applications, but also shapes their architectures. For clarity purposes it is assumed that the architectures follow a tree topology. In reality, this is not necessarily true as in the case of the RD-IM the nodes form a complete graph, and for the RSD-IM nodes inside the s_groups, they are also complete graphs.

Deployment follows an strict sequence determined by the supervision tree. This is logical as those processes with supervision responsibilities are in charge of spawning their children processes. Hence, unless explicitly stated, these steps are the same for RD-IM and RSD-IM:

1. *launcher* module determines the total number of *router* processes and the *server_supervisor* processes that are children of each of these *router* processes. Then it creates the *router_group* s_group (RSD-IM only)

2. For each router node in the architecture, *launcher*

   - adds the node to the *router_group* s_group (RSD-IM only),
   - creates a *server_group* s_group and adds the router and all children server nodes to it (RSD-IM only),
   - spawns one *router_supervisor*.

3. This *router_supervisor*

   - spawns its own monitor,
   - spawns the relay processes (RSD-IM only)
   - spawns its children *router* processes.

4. Each router process then determines its *server_supervisor* children processes, and spawns them sequentially.

5. Finally, each *server_supervisor* spawns the corresponding *DB* processes.

When all the server-side processes are launched, the module spawns the *routers_DB* processes in the client nodes (cf. section 2.2.2).

Figures 2.2 and 2.3 depict the resulting architectures after the deployment of the applications. These figures also show the simplification of the architecture introduced by SD-Erlang. RD-IM uses the default distribution model provided with Erlang: all the connections between *server-side* (i.e., *server* and *router*) nodes are transitive. In the case of SD-Erlang, these transitive connections are established only inside the s_groups. In both applications, client nodes connections are established as *hidden*.

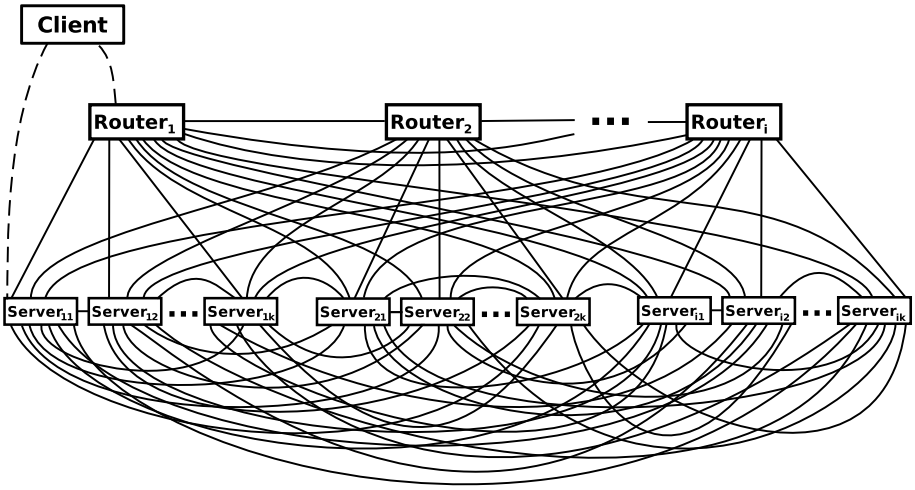

Figure 2.2: Connections in RD-IM architecture (client connections not shown).

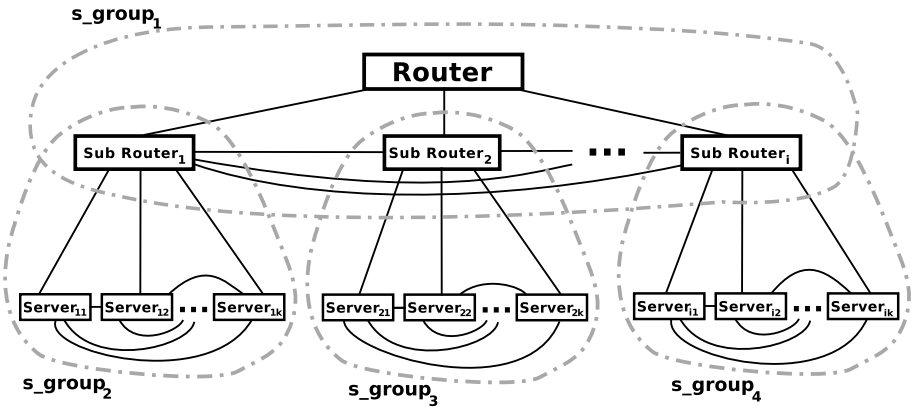

Figure 2.3: Connections in RSD-IM architecture (client nodes not included).

2.5 Auxiliary Non-IM modules used for testing purposes

It is worth of mention two modules that are used to test reliability and to record the measurements of the benchmarks: the *rhesus* module and the *logger* module.

Rhesus Module. Erlang approaches the error handling according to the *"let it crash"* principle. Applications are written in two parts: one that performs the operations required to solve the problem, and a second that corrects errors as soon as they arise. This strategy brings several advantages such as (i) lesser defensive code, (ii) focusing on the recovery of the system rather than preventing the errors, or (iii) better diagnosis of the failures.

One of the assumptions made after this *"let it crash"* principle is that applications must be able to provide (at least some) functionality under a failure scenario. It was in order to test whether this assumption held for their Amazon Web Services, that the engineers at Netflix developed the tool Chaos Monkey (Bennett and Tseitlin, 2012): ¡¡*Chaos Monkey is a service which runs in the Amazon Web Services (AWS) that seeks out Auto Scaling Groups (ASGs) and terminates instances (virtual machines) per group*¿¿ (*íbid.*).

Chaos Monkey is a tool of great interest within the context of the RD-IM application for obvious reasons. However, the Netflix version does not quite fit the requirements as it is *node-focused*.

There is an alternative implementation of this tool credited to Daniel Luna[2] that is more pertinent to the IM. Luna's Chaos Monkey differs from the original in that (i) it is written in Erlang/OTP instead of Java, and (ii) it terminates instances of processes rather than instances of virtual machines (nodes). Unfortunately, as the RD-IM application is not using standard OTP behaviours, this version of the tool turned out to be incompatible.

For this reason, it was decided to build a custom-made module that shows a similar behaviour to Chaos Monkey. This module, named Rhesus after the rhesus macaque, has the following features:

- Random termination time for a process ranging from five seconds to one hour.
- User-defined termination time for a process.
- Weighted termination probability of router/server processes.
- Weighted termination probability of the server processes.

[2] https://github.com/dLuna/chaos_monkey

The basic principle behind this chaos generation scheme is quite simple. Rhesus looks for a router_supervisor
_process and injects termination messages into the architecture. The time rate at which these messages are sent is either randomly picked at the very first cycle of its execution, or is passed as a parameter at the call of the function `chaos_on/1`.

The sequence of steps that rhesus follows to terminate a process is rather simple:

1. Rhesus chooses to finish either a server process or a router process. This pick is random, yet the probabilities are 1:4 that a router process is chosen (hence, 3:4 for the server processes).

2. It sends the appropriate message to the *router_supervisor* process.

3. If the *router_supervisor* receives the instruction to terminate a router process, it executes an `exit/2` passing as the argument the pid of a selected router process (either *router_process*, *router_supervisor* process, or *router_supervisor_monitor* process). The probabilities are, in this case, 1:*number of router processes + 2*.

4. If the *router_supervisor* receives the order of terminating a server process, then it forwards the request to a *router_process* as if it were a *log in* request, for example.

5. The *router_process* takes an aleatory pid of a server, and sends the termination request to be processed.

6. When the process termination request arrives, the *server_supervisor* process deals the issue in a similar way to how the router supervisor does. In this case the probabilities are 1:5 for *monitored_processes_DB*, *Chat_DB*, *Clients_DB*, and *server_supervisor* processes, and 1:10 for *client_monitor* and *chat_session* processes.

This chaos generation logic exploits the fact that there are data structures that store the pids of the IM processes running on each node. This makes the implementation fairly straight forward, making unnecessary to refactor the whole application using OTP behaviours. Besides, the rhesus module offers an additional advantage compared to Luna's application: it sets *a priori* probabilities for the termination of the processes.

The Erlang/OTP Chaos Monkey does not make any distinction amongst the different types of process that conform the application. In practical terms this means that all the processes have the same chances to be terminated. Since the most frequent processes in the IM are client monitors and chat sessions. If all the processes have the same probability of being finished, it would be very unlikely to find an scenario in which the elected process

to stop is not one of those two. Or, similarly, it would be necessary to run extraordinarily long experimental series to make sure that other processes than client monitors and chat sessions are stopped. And this is something not convenient for the present case.

Logger Module. This module provides a measurements-gathering facility, that is useful to study the performance of the IM applications in terms of latency and throughput.

The module provide a set of *recorder* functions that are spawned at the client nodes whenever data-collection is required. At launch time, they create a comma separated value (.csv) file to which they write the latency or throughput data in real-time. When the data collection session has finished, they just close the file and if there are no more sessions left they terminate.

Recorders can be customised, allowing to specify the name and path of the output files, the number of series to be recorded, and the length of the data collection sessions. For throughput measurements, they also allow to set a threshold in latency that can be used to determine the quality of the service (defined as percentage of messages delivered below threshold).

2.6 Summary

This chapter reviews the implementation, hierarchy and architecture of the instant messenger applications. The design of the application follows the supervision tree principle imposed by the Erlang standards for reliable applications. Even though behaviours have not been used, process monitoring is performed.

The supervision tree imposes a hierarchy to both nodes and processes. It also guides the deployment sequence of the applications.

On the other hand, the resulting architectures after deployment are of a very different complexity. This is the result of the application of SD-Erlang technology.

Finally, two auxiliary modules have been described. Yet they do not belong to the IM applications themselves, they play a crucial role in project: they provide the means for data-collection and reliability testing.

Chapter 3

Reliability Mechanisms and Message Flow

This appendix includes the sequence diagrams that model the normal operation of the instant messengers, as well as the reliability mechanisms. These were used as a guide for the implementation.

3.1 D-Erlang Instant Messenger

The reliability of the system is achieved by the implementation of a series of monitoring processes called ***supervisors***. In the case of the D-Erlang architecture there are (at least) two of these supervisor processes: (at least) one at router node and (at least) another one at server node.

There are several situations in which the system can behave in an undesirable manner, mostly due to bad calls, or processes that finish for reasons other than a normal exit call sent to a process to finish. In these cases, the supervisors must recover the system, taking it again to a valid state. Some of these misbehaviours of the system and how the supervisors tackle them are detailed below.

3.1.1 Handling Exceptions

A client tries to log in to the system when it is logged in already

1. Router *process* forwards the request to *server_supervisor*.
2. *server_supervisor* then spawns a new *client_monitor* process.
3. This new *client_monitor* queries the *Client_DB* to check that the client is not already registered in the system.
4. *Client_DB* then sends a message to the new *client_monitor* informing that the client is already logged in.

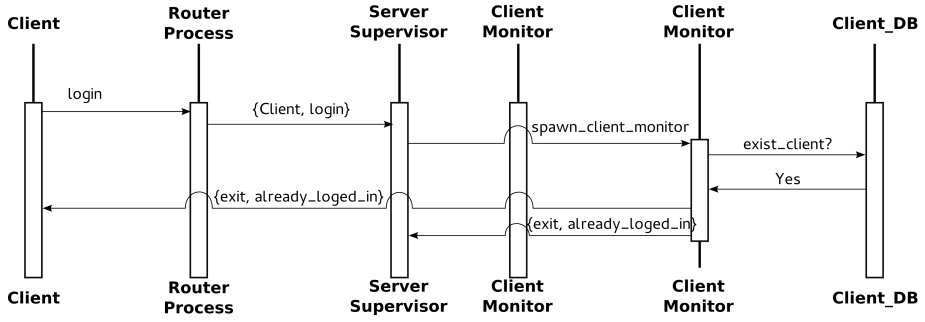

Figure 3.1: Illegal login sequence diagram.

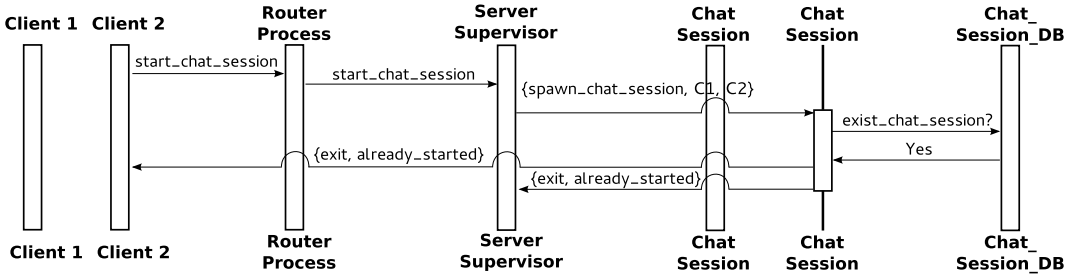

Figure 3.2: Illegal chat session sequence diagram.

5. Finally, the newly created *client_monitor* notifies the client and *server_supervisor*, and terminates.

A client tries to establish a chat session with another client to whom it already has an opened chat session

1. *server_supervisor* process spawns a new *chat_session* process.
2. This newly created chat session queries the *Chat_Session_DB* to ensure that it does not exist already.
3. The *Chat_Session_DB* sends a message to the new *chat_session* process indicating that it exists already.
4. The new *chat_session* process then notifies the error to both: the client who requested the chat session, and the *server_supervisor*, and terminates.

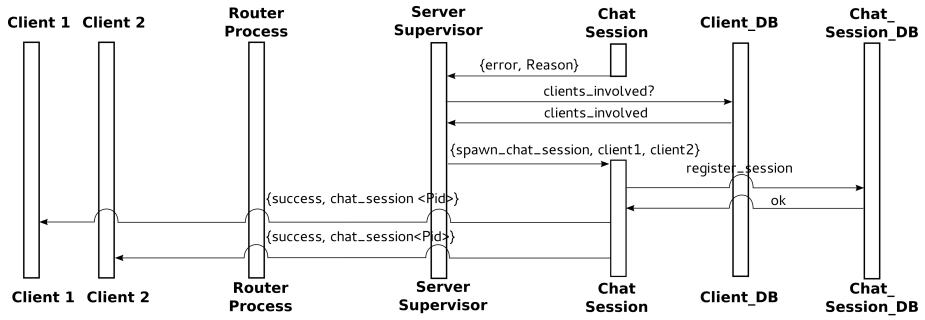

Figure 3.3: Chat session recovery from crash sequence diagram.

chat_session process crashes before the session is finished

1. A system error message {`error, Reason`} is sent to *server_supervisor*.
2. As soon as *server_supervisor* traps this message, it works out what clients were involved in this faulty chat session. A query to *Chat_Session_DB* including the `Pid` of the faulty process trapped in the message should be enough to do so.
3. Next, it deletes this crashed session from the *Chat_Session_DB*.
4. Finally, the *Server_Supervisor* spawns a new *Chat_Session* process using the information gathered, and
5. notifies the clients the `Pid` of the new chat session process.

server_supervisor process crashes

1. *router_supervisor* receives a system error message {`error, Reason`}, and as soon as this error is trapped,
2. the *router_supervisor* spawns a new *server_supervisor* process.
3. The new *server_supervisor* requests the active sessions and clients to the *Chat_Session_DB* and *Clients_DB*.
4. As soon as the new *server_supervisor* collects the information about the pertinent clients and sessions, it starts monitoring the appropriate *client_monitor* and *chat_session* processes.

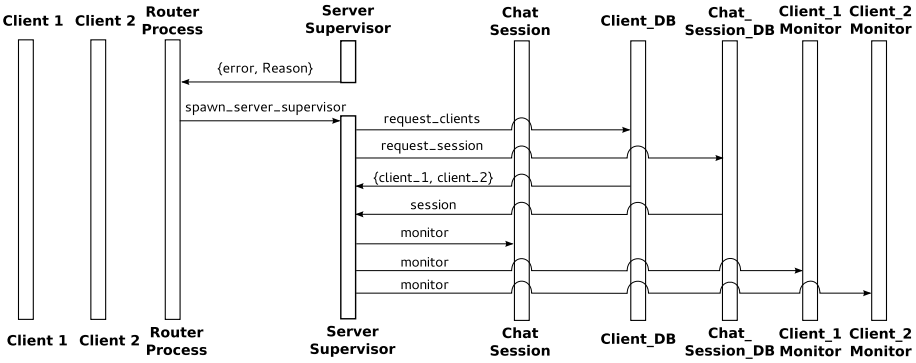

Figure 3.4: Server supervisor recovery sequence diagram.

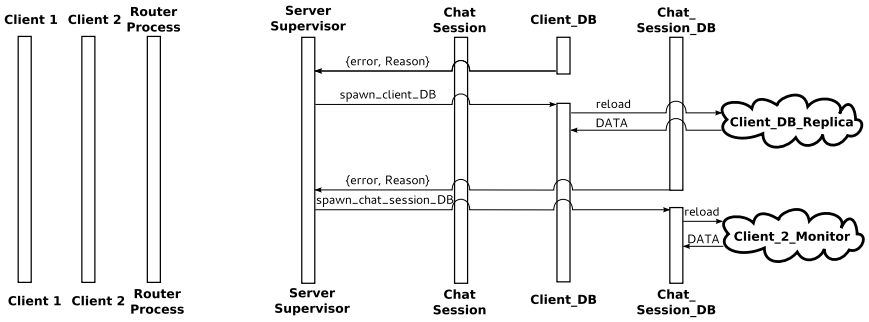

Figure 3.5: Databases recovery sequence diagram.

***Chat_Session_DB* or *Client_DB* fails** In this case, the courses of action are the same for both, thus we will just indicate *DB* to refer either *Chat_Session_DB* or *Client_DB*.

1. System sends a an error message to the *server_supervisor*.
2. The supervisor traps this error and spawns a new *DB* process.
3. Once the *DB* is started, it fetches the data from a replica, to get into the same state as it was before it crashed.

3.1.2 Normal Behaviour

On the other hand, during the normal execution of the system there is also a constant flow of messages that characterises the behaviour of the instant messenger:

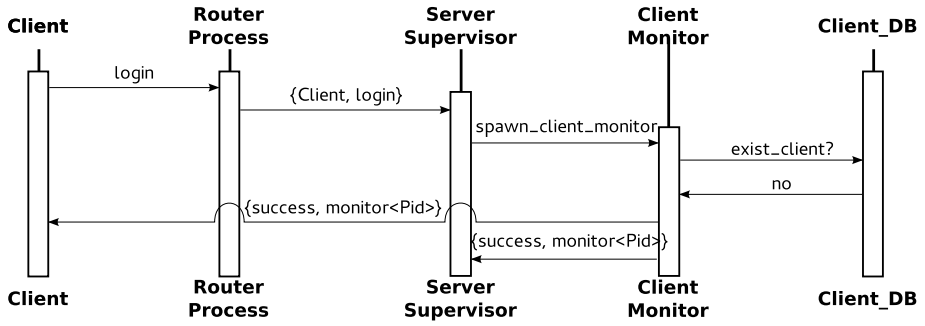

Figure 3.6: Login sequence diagram.

Client login

1. Client sends a *login request* message to *router* process.
2. *router* process selects a server node and forwards the request to the corresponding *server_supervisor* process.
3. This *server_supervisor* spawns a *client_monitor*, and
4. this *client_monitor* queries the *Client_DB* to find out whether the client is already logged in or not.
5. If the client is not logged in yet, then *Client_DB* informs the *client_monitor* that the client is not logged in. Then *client_monitor* registers the client in the *Clients_DB*, and
6. notifies the client that login has been successful.

Chat session

1. Client_1 sends a `start_chat_session` message to *router* process, who
2. forwards the request to the *server_supervisor*;
3. this supervisor then spawns a *chat_session* process.
4. The created *chat_session* process queries the *Chat_Sessions_DB* to ensure that it does not exist already.
5. In normal circumstances, *Chat_Sessions_DB* answers that the session does not exist, and
6. *chat_session* process, informs the involved clients the success of the operation.

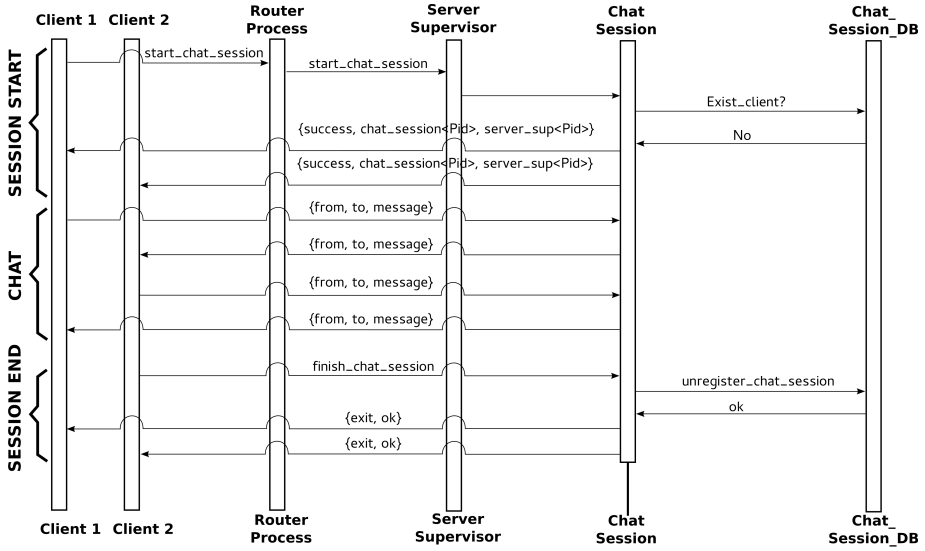

Figure 3.7: Chat session sequence diagram.

7. Once the *chat_session* is established, the clients send messages to each other.

8. To finish a chat session, one of the clients sends a `finish_chat_session` message to the *chat_session* process.

9. *chat_session* removes the chat session from the *Chat_Sessions_DB*, and right after that

10. it notifies the clients.

11. Finally, *chat_session* process sends a {`exit, session_finished`} message to the server supervisor, and terminates.

Client logout

1. Client sends a *logout request* message to its *client_monitor* process.

2. When *client_monitor* traps the request, it deletes the client from the *Client_DB*,

3. and notifies the client that logout has been successful.

4. Finally, *client_monitor* terminates.

In this case, the client has not active chat sessions when it logs out. However, if the client crashes or logs out while chatting, the sequence of events is slightly different. In this case,

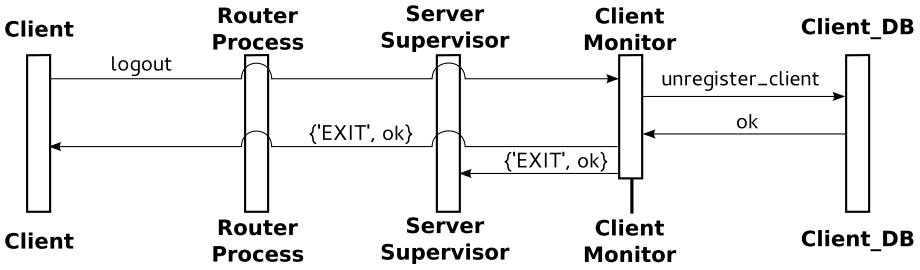

Figure 3.8: Logout sequence diagram.

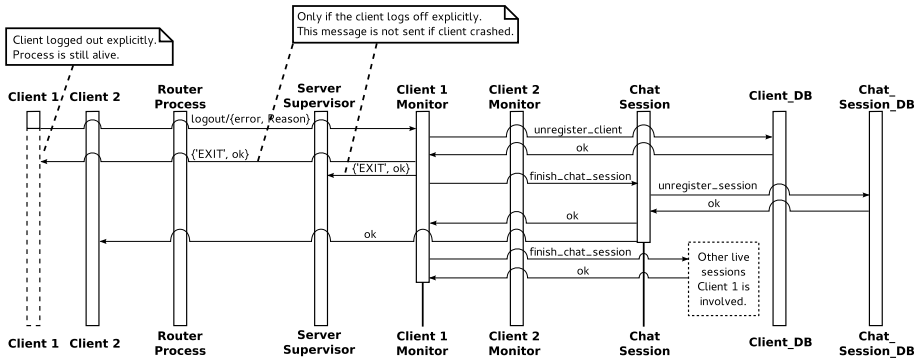

Figure 3.9: Logout sequence diagram.

1. the *client_monitor* receives either a logout request or an {`error, Reason`} message. In both cases
2. *client_monitor* unregisters the client from the *Client_DB*.
3. Right after the client is removed from the database, the *client_monitor* sends a `finish_chat_session` request to the *chat_session* process, which would finish the session in a normal way.
4. Finally, the *client_monitor* traverses all the entries in the *Chat_Session_DB* to discover all the sessions the client was involved, and finish them sendind the corresponding `finish_chat_session` message (cf. *chat session* description).

3.2 SD-Erlang Instant Messenger

As in the case of the D-Erlang instant messenger system (cf. Section 3.1), the SD-Erlang version of the system has also different mechanisms to ensure the

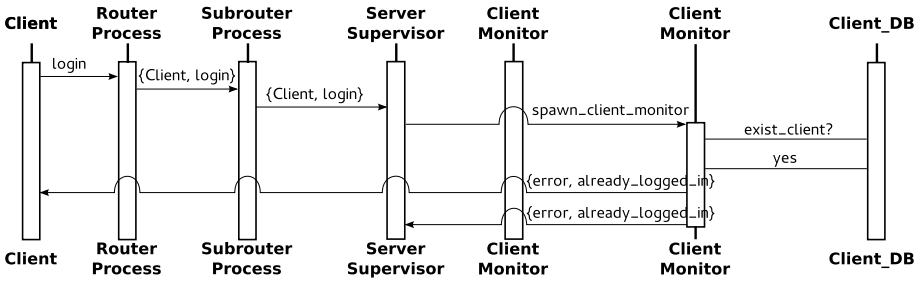

Figure 3.10: Illegal login sequence diagram.

reliability of the system. This reliability is achieved, as in the previous case, implementing monitoring processes called ***supervisors***. The SD-Erlang architecture contains (at least) three of these supervisor processes: (at least) one at *router* node, a second one (at least) at *subrouter* node, and (at least) another one at *server* node.

The following paragraphs describe different situations in which the normal operation of the system is compromised, and the ways the system recovers from this abnormal behaviour:

1. *A client tries to log in to the system when it is logged in already.*
   (a) Router *process* forwards the request to *subrouter_process*.
   (b) The *subrouter_process* then forwards the request to the *server_supervisor*,
   (c) and this *server_supervisor* spawns a new *client_monitor* process.
   (d) This new *client_monitor* queries the *Client_DB* to check that the client is not already registered in the system.
   (e) *Client_DB* then sends a message to the new *client_monitor* informing that the client is already logged in.
   (f) Finally, the newly created *client_monitor* notifies the client and *server_supervisor*, and terminates.

2. *A client tries to establish a chat session with another client to whom it already has an opened chat session.*
   (a) *server_supervisor* spawns a new *chat_session* process.
   (b) This newly created chat session queries the *Chat_Session_DB* to ensure that it does not exist already.
   (c) The *Chat_Session_DB* sends a message to the new *chat_session* process indicating that it exists already.
   (d) The new *chat_session* process then notifies both: the client who requested the chat session, and the *server_supervisor*, and terminates.

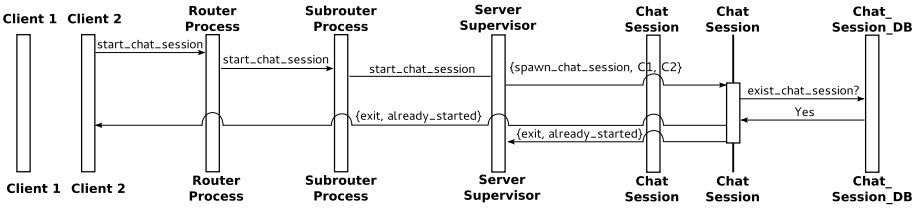

Figure 3.11: Illegal chat session sequence diagram.

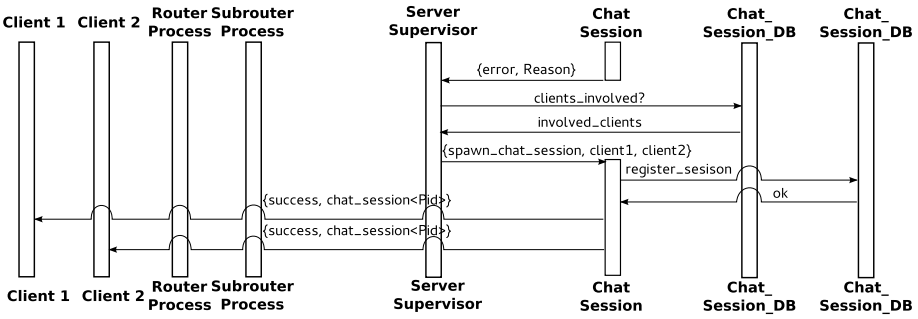

Figure 3.12: Chat session recovery from crash sequence diagram.

3. chat_session *process crashes before the session is finished.*

   (a) A system error message {`error, Reason`} is sent to the *server_supervisor* process.

   (b) As soon as this *server_supervisor* traps the error message, it works out what clients were involved in this faulty chat session. A query to *Chat_Session_DB* including the PID of the faulty process should be enough to do so.

   (c) Next, this crashed session is removed from the *Chat_Session_DB*.

   (d) Finally, the *Server_Supervisor*, spawns a new *Chat_Session* process with the information gathered, and

   (e) notify the clients the new Pid of the chat session process.

4. router_process *crashes*

   (a) *router_supervisor* traps the error message {`error, Reason`}, and

   (b) query the *supervisor_DB*, the *Supervisor_DB* to fetch the Pid of the process supervised by the failed *router_process*.

   (c) Finally, the *router_supervisor* spawns a new *router_process*, and

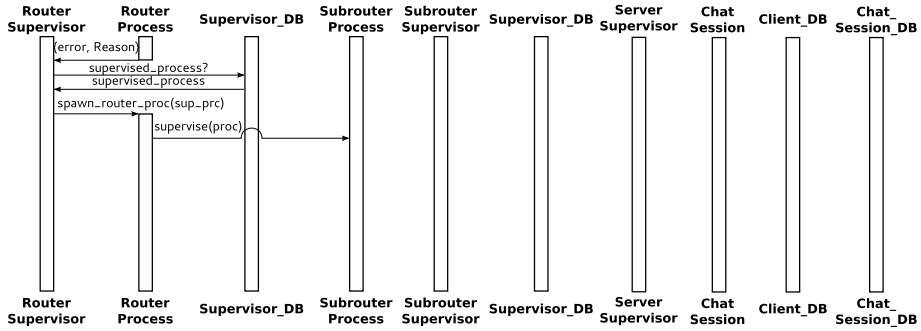

Figure 3.13: Router process recovery sequence diagram.

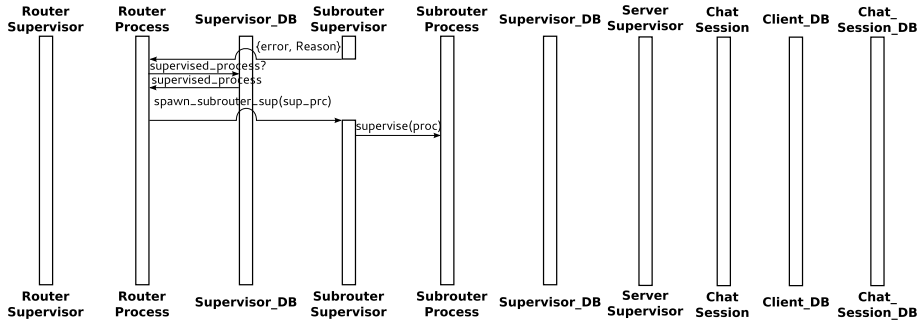

Figure 3.14: Subrouter supervisor process recovery sequence diagram.

(d) this new spawned process is linked to the *subrouter_process* that was supervised by the failed process.

5. subrouter_supervisor *process crashes*

   (a) *router_process* receives an {`error, Reason`} message from the failed *subrouter_supervisor*.

   (b) As soon as this error message is trapped, the *router_process* queries the *supervisor_DB*, to retrieve the Pid of the *subrouter_supervisor* process monitored by the failed *subrouter_supervisor*.

   (c) As in the previous case, a new *subrouter_supervisor* is spawned, and

   (d) linked to the *subrouter_process* that was supervised by the failed process.

6. subrouter_process *crashes*

   In this case, the recovery logic is much the same as when a *router_process* crashes:

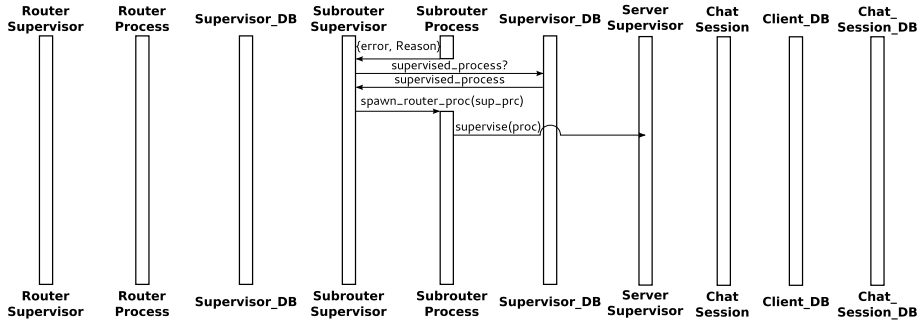

Figure 3.15: Subrouter process recovery sequence diagram.

(a) *subrouter_supervisor* traps the error message {`error, Reason`}, and

(b) query the *supervisor_DB*, the *Supervisor_DB* to fetch the Pid of the process supervised by the failed *subrouter_process*.

(c) Again, the *subrouter_supervisor* spawns a new *subrouter_process*, and

(d) this new spawned process is linked to the *server_supervisor* that was supervised by the failed process.

7. server_supervisor *process crashes*

   (a) *subrouter_process* receives a system error message {`error, Reason`}, and as soon as this error is trapped, the *subrouter_process*

   (b) spawns a new *server_supervisor* process.

   (c) the new *server_supervisor* requests the active sessions and the clients to the *Chat_Session_DB* and *Clients_DB*.

   (d) As the new *server_supervisor* collects the clients and sessions, it starts monitoring the corresponding *client_monitor* and *chat_session* processes.

8. Chat_Session_DB *or* Client_DB *fails*. In this case, the courses of action are the same for both, thus we will just indicate *DB* to refer either *Chat_Session_DB* or *Client_DB*. This schema is also applied to the *Supervisor_DB* data structures metioned in cases 4, 5, and 6. We will not describe the case to not overload the reader with redundant information.

   (a) System sends a an error message to the *server_supervisor*.

   (b) The supervisor traps this error and spawns a new *DB* process.

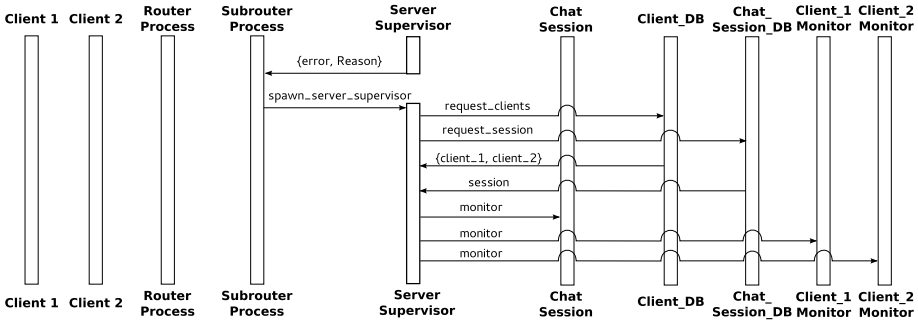

Figure 3.16: Server supervisor recovery sequence diagram.

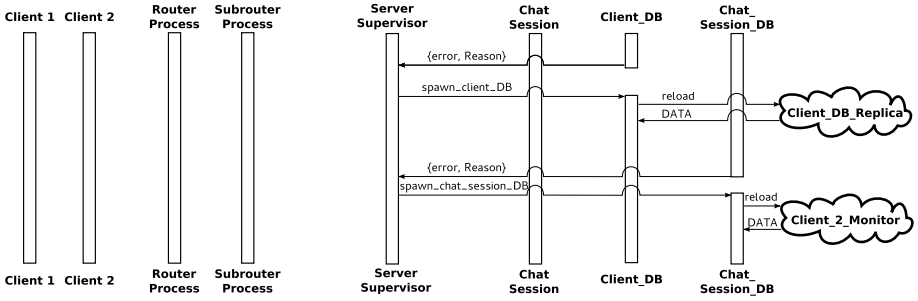

Figure 3.17: Databases recovery sequence diagram.

(c) Once the *DB* is started, it fetches the data from a replica, to get into the same state as it was before it crashed.

On the other hand, during the normal execution of the system there is also constant a flow of messages that characterises the behaviour of the instant messenger:

1. *Client login.*
   (a) Client sends a *log-in request* message to *router_process.*
   (b) The *router_process* forwards this message to a *subrouter_process* that
   (c) forwards the request to the corresponding *server_supervisor* process.
   (d) This *server_supervisor* spawns a *client_monitor*, and
   (e) this *client_monitor* queries the *Client_DB* to find out whether the client is logged in or not.

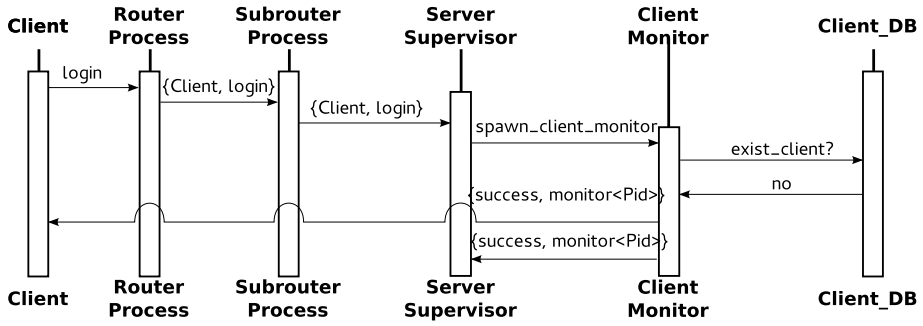

Figure 3.18: Login sequence diagram.

(f) If the client is not logged in yet, then *Client_DB* informs the *client_monitor* that the client is not logged in. Then *client_monitor* registers the client in the *Clients_DB*, and

(g) notifies the client that login has been successful.

2. *Chat session.*

(a) Client_1 sends a `start_chat_session` message to *router* process, who

(b) forwards the request to a *subrouter_process*, which in turn will

(c) forward the request to a *server_supervisor*. This supervisor spawns a *chat_session* process.

(d) The created *chat_session* process queries the *Chat_Sessions_DB* to ensure that it does not exist already.

(e) In normal circumstances, *Chat_Sessions_DB* answers that the session does not exist, and

(f) *chat_session* process, informs the involved clients the success of the operation.

(g) Once the *chat_session* is established, the clients send messages to each other.

(h) To finish a chat session, one of the clients sends a `finish_chat_session` message to the *chat_session* process.

(i) *chat_session* removes the chat session from the *Chat_Sessions_DB*, and right after that

(j) it notifies the clients.

(k) Finally, *chat_session* process sends a {`exit, session_finished`} message to the server supervisor, and

(l) terminates.

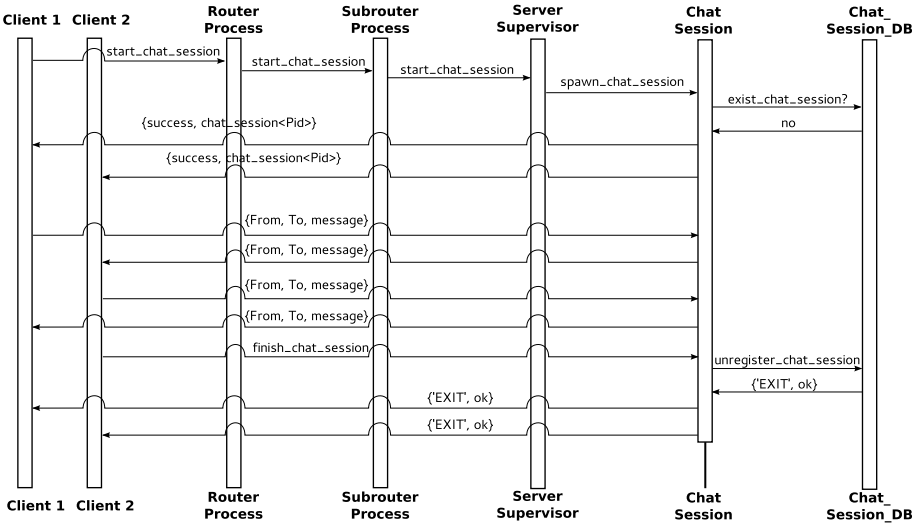

Figure 3.19: Chat session sequence diagram.

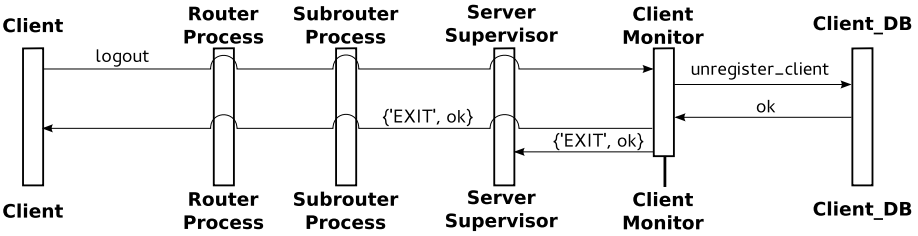

Figure 3.20: Logout sequence diagram.

3. *Client logout*

   (a) Client sends a *log-out request* message to *client_monitor* process.

   (b) When *client_monitor* traps the request, it deletes the client from the *Client_DB*,

   (c) and notifies the client that logout has been successful.

   (d) Finally, *client_monitor* terminates.

   In this case, the client has not active chat sessions when it logs out. However, if the client crashes or logs out while chatting, the sequence of events is slightly different. In this case,

   (a) the *client_monitor* receives either a logout request or an {`error, Reason`} message. In both cases

   (b) *client_monitor* unregisters the client from the *Client_DB*.

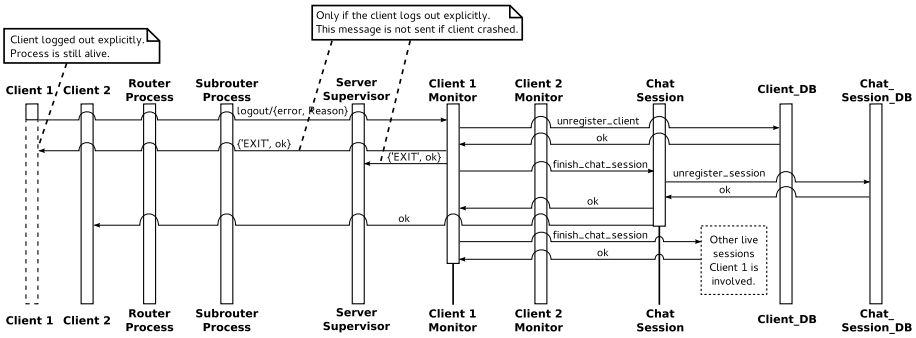

Figure 3.21: Logout sequence diagram.

(c) Right after the client is removed from the database, the *client_monitor* sends a `finish_chat_session` request to the *chat_session* process, which would finish the session in a normal way.

(d) Finally, the *client_monitor* traverses all the entries in the *Chat_Session_DB* to discover all the sessions the client was involved, and finish them sendind the corresponding `finish_chat_session` message (cf. *chat session* description).

Chapter 4

Changes to the Original Design

The implementation of the system tries to fulfil the different behaviours described on the section 3.1 *Reliability mechanisms and messages flow*. However, due to the constraints imposed by the language, and decisions taken at implementation time, the final system does not reproduce exactly the actions depicted on this document. This section enumerates such changes, while also offering some additional information.

1. *A client tries to log in to the system when it is logged in already.*

   When a client, logs in to the system, it is also locally registered in the node in which is executing. Hence, if the client tries to log in again in that node, the first thing it does is to check whether there is a registered process in that node with the same name. If that is the case, then the login action is aborted and an error message informs of this situation. If the client tries to log in from another node, then the sequence described in section 3.1 is executed.

2. *A client tries to establish a chat session with another client to whom it already has an opened chat session.*

   The final implemented version does not make any explicit `start_session` calls to start a chat session. That is, the client does not have to perform a chat session request before being able to send any messages to another client.

   Client processes keep track of the chat sessions they are participating. Every time a client receives {`Sender, Receiver, Message, Send_Message`}, it tries to retrieve the *pid* of that chat session from their internal opened sessions storage.

   If there is not such chat session, the client sends a chat session request to the *server_supervisor* process. The *server_supervisor* spawns the

chat_session process, returns the *pid* to the client, and then the client sends the message. If that session exists already, the client sends the message to the corresponding *chat_session* process. Therefore, this situation cannot happen.

3. chat_session *process crashes before the session is finished*

   This is probably the situation where the original design differs the most respect to the final implementation. Due to the way in which the chat sessions are started (i.e., they do not have to be set up explicitly), there is no need of handling this error and recover the process: when the next message is sent, a new *chat_session* will be spawned. For that reason, the recovery strategy consists of behaving as if the *chat_session* process finished normally.

4. server_supervisor *process crashes.*

   When the *router_process* traps an abnormal termination of the *server_supervisor*, it only knows the *pid* of this crashed process. Even though it is possible to access the the *client_db* and *chat_db* processes –as they are globally registered processes– it was decided to keep a record of the monitored processes by the *server_supervisor* on a separate *monitored_db* process to make the recovery sequence faster. This *monitored_db* process is similar to the other *db* processes, but it is registered locally inside the server node.

   The recovery strategy involves the spawning of a new *server_supervisor* process, and the traversal of this *monitored_db*, instead of traversing the client and chat databases to retrieve the required information. In fact, this is actually the strategy proposed for the recovery of the *router_process*, *subrouter_supervisor* and *subrouter_process* processes in the SD-Erlang version of the system (cf. section 3.2).

5. *Chat_Session_DB* or *Client_DB* fails.

   There was introduced an improvement, respect to the original strategy. If the crashed database is the main, during the recovery strategy, main and replica databases are swapped. The reason is that if any other process needs to access the information stored while the database is recovering, it is available since the swapping requires only one operation.

6. *Client login*

   Changes at this level have been pointed out already. The original proposal assumed two explicit operations:

   (a) Start the client process.

   (b) Execute the login.

The actual implementation only requires the second. When the user executes the function

```
client:login(atom()),
```

this function starts a client which *self-logs in* following the sequence described in Figure 3.6.

7. *router_supervisor* crashes.

   The original reliability model did not contemplate this failure. The most sensible way to tackle this situation is to make a ¡¡*keep_alive*¿¿ process. However, after facing several problems, a less efficient but similarly effective tactic was implemented. At launch time, *router_supervisor* spawns and monitors a sister process, whose function is only to monitor the *router_supervisor*. This *router_supervisor_monitor* process keeps the same information as the monitored process, at any time. Thus when the *router_supervisor* finishes unexpectedly, the monitor process spawns a new process, feeding the information of the processes that need to be monitored.

   When the *router_supervisor_monitor* crashes, the *router_supervisor* spawns a new one, feeding the information into the newly spawned process.

8. *client_process* crashes.

   Since the client side is not of the interest of this research, section 3.1 does not include the case of the unexpected termination of client processes. However, this circumstance also has effects at the server side. In particular, it is necessary to terminate the chat sessions in which this crashed client is taking part, as well as remove the client from the *Clients_DB*. Thus, *client_monitor* also traps these abnormal termination of client processes behaving as if the client has logged out normally.

9. *SD-Erlang Router/Subrouter* processes. The original design included a layer of routers and then a second layer of slave routers that would be in charge of monitoring the servers. At the moment of the implementation it was seen that the top layer of routers did not any advantages and, potentially it could be a bottleneck. For this reasons, the final implementation lacks of this extra layer of routers.

Chapter 5

Evaluation

This chapter describes the results of the tests made on the instant messenger applications.

5.1 Experimental Setup

All the measurements have been taken using the GPG cluster located at the University of Glasgow School of Computing Sciences. This is a Beowulf cluster composed of 20 nodes, of which 17 are used. The reason is that there are three nodes that can be used by any member of the school without prior notice, and this circumstance can contaminate the results obtained.

Each of the nodes in the cluster has the following configuration:

- 16 cores (2×Intel Xeon E5-2640 2 GHz).
- 64 GB RAM (4 GB RAM/core).
- 300 GB local disk.
- 10 Gb Ethernet interconnect

The software configuration of these nodes is as follows:

- Scientific Linux 6 (Carbon) 64-bit.
- Erlang/OTP 17.4 (modified by the RELEASE project), downloaded and built from the RELEASE Github repository.

5.2 Experiment 1: Impact of the Number of Servers with No Failures

Design. In the first experiment we analyse whether there are differences in throughput between the Distributed Erlang version of the instant messenger

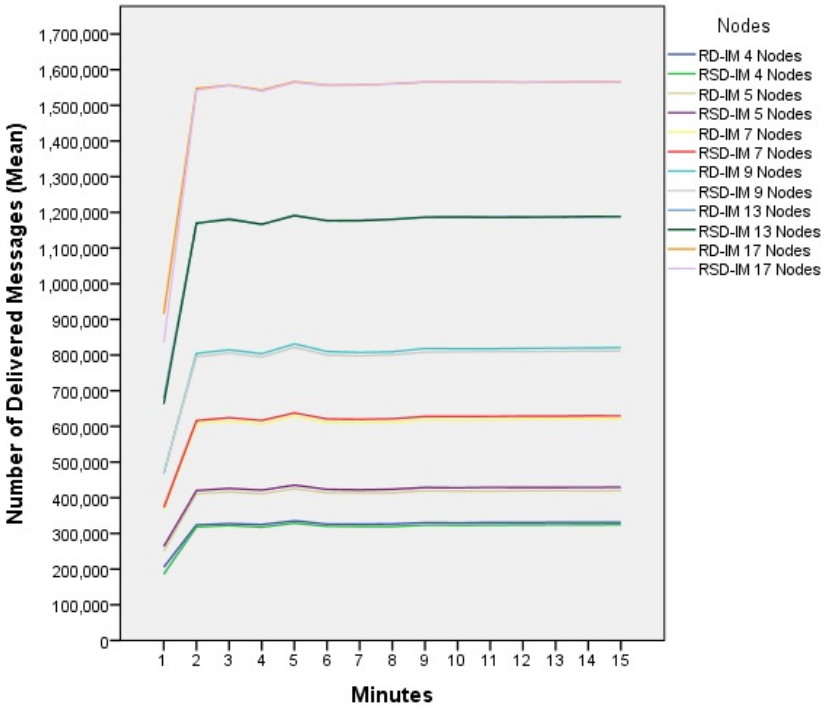

Figure 5.1: D-IM vs SD-IM. Maximum throughput time series.

(D-IM) and the Scalable Distributed Erlang version of the instant messenger (SD-IM). For that we gradually increase the number of server nodes (3, 4, 6, 8, 12, 16) while keeping the number of router nodes fixed – equal to one. The reliability is not considered in this experiment

This setup results identical architectures for both IM applications, since the SD-IM only has one *s_group*. Hence, *s_group* operations in the SD-IM are analogous to the global operations in the D-IM. Traffic is injected at same rates in all conditions. Traffic generation is modelled after [XGT07].

Results. Figure 5.1 shows the evolution of the throughput for all the architectures considered, in time series of 15 minutes each. It can be observed that it takes about six minutes for the throughput to become stable. This can be due to the implementation of the traffic generation logic. Traffic is started by the traffic generators in a big initial burst. Despite the fact that messages are sent at random times between one and 20 seconds, during the first minutes messages may be overloading the queues of processes at the server nodes (e.g., *server_supervisor* or *chat_db* processes). As chat sessions are taking place, the traffic becomes more uniform and the size of these queues diminishes.

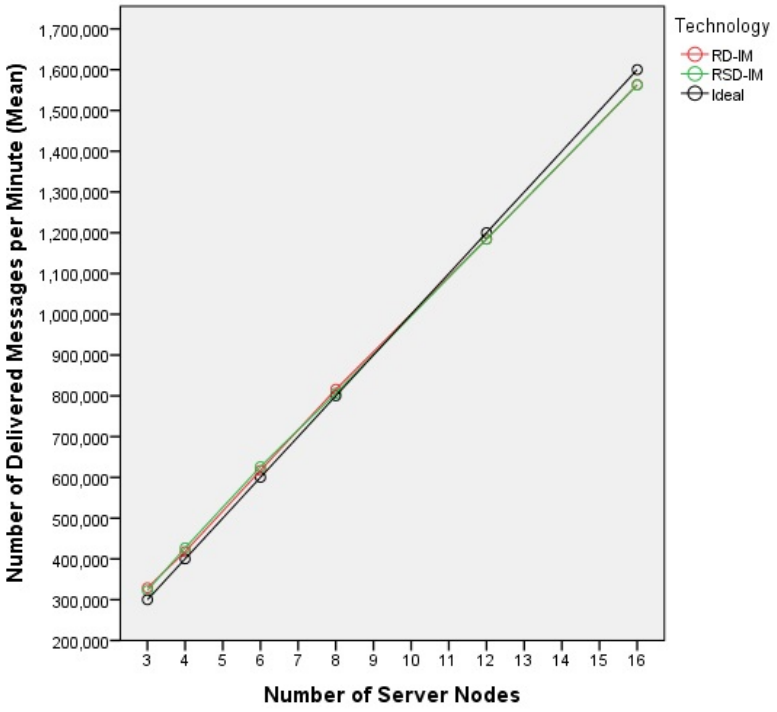

Figure 5.2: D-IM vs SD-IM. Maximum mean throughput when applications are stable.

Figure 5.1 also show the way the IM applications scale. Considering that the Y-axis show the number of messages per minute that go through the architecture, the first evident fact is that the differences between technologies (D-Erlang and SD-Erlang) are minimal, if any. More interesting, though is the linear relationship between the number of messages per minute and the number of server nodes present. This figure suggests that these *server nodes have an effective capacity of handling approximately* 10^6 *messages per minute*. In fact, the mean values of the delivered messages when the applications are stable seem to reinforce this ratio (Figure 5.2).

A closer inspection of Figure 5.2 shows that the number of messages that each of these server nodes is able to handle decreases as the number of server nodes increases. Hence, in architectures up to eight server nodes, these handle slightly more than the already mentioned 10^6 messages. Conversely, in larger architectures the server nodes are able to handle slightly fewer messages than 10^6.

Considering the architectures tested altogether, an independent samples test indicates that the mean throughput of the SD-IM (916,234 messages/minute) is larger than the throughput of the D-IM (765,351 mes-

sages/minute; p <0.5). However, this result must be handled carefully. This same independent samples test applied to each of the different architectures considered, shows that SD-IM has an statistically significant higher throughput than D-IM only for those architectures comprised of four and six server nodes. Since for the rest of the conditions throguhputs seem to be the same, it is safer to assume that *there are no performance differences between the applications*, and that these differences must be caused for some unspecific random factor.

5.3 Experiment 2: Impact of the Number of Servers & S_groups with No Failures

Design. The second experiment analyses the differences in throughput between the D-IM and the SD-IM when the number of servers varies, but the number of routers is kept constant. That is a 2×3 factorial design with factors *technology* –namely, Distributed Erlang and Scalable Distributed Erlang— and *number of server nodes* –with three levels: 6, 8, and 12.

The number of router nodes was fixed to two. This allows the creation of two *s_groups* in the SD-IM that vary in size according to the number of server nodes present in each of the experimental conditions. Therefore, the *s_groups* in the case of the SD-IM were composed of one router and three, four or six servers, depending on the level of the variable *number of server nodes* under test. As in the case of Experiment 1, reliability is not considered in this experiment.

Results. Figure 5.3 shows the evolution of the throughput in the different configurations of the instant messenger architectures through a time series of 15 minutes. As in the case of the previous experiment it takes approximately 6 minutes for the applications to become stable. Once the applications reach stability, they show constant differences in throughput only in the experiment with two router and eight server nodes architecture.

When the number of server nodes is set to eight, these differences are statistically significant (p <0.05) although the throughput of SD-IM is only a 2% higher than the throughput of D-IM. In turn, when the number of servers is six and 12 these differences in throughput disappear (p >0.05).

These later results seem to indicate that the differences found in the case of eight server nodes might be caused for some external factor such as the state of the cluster when the tests were running. In this sense, Figure 5.4 shows more clearly this peak in the throughput of SD-IM when the number of server nodes is eight. However, the overlapping of the mean throughputs in the other two considered conditions, reinforce the idea that in the difference found is due to an external factor.

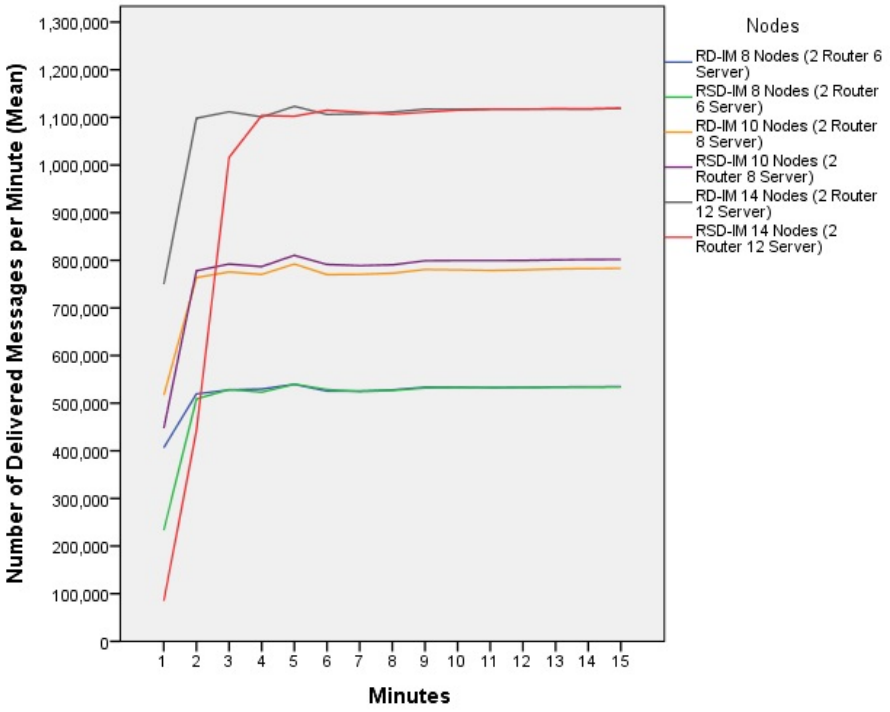

Figure 5.3: D-IM vs SD-IM. Maximum throughput time series.

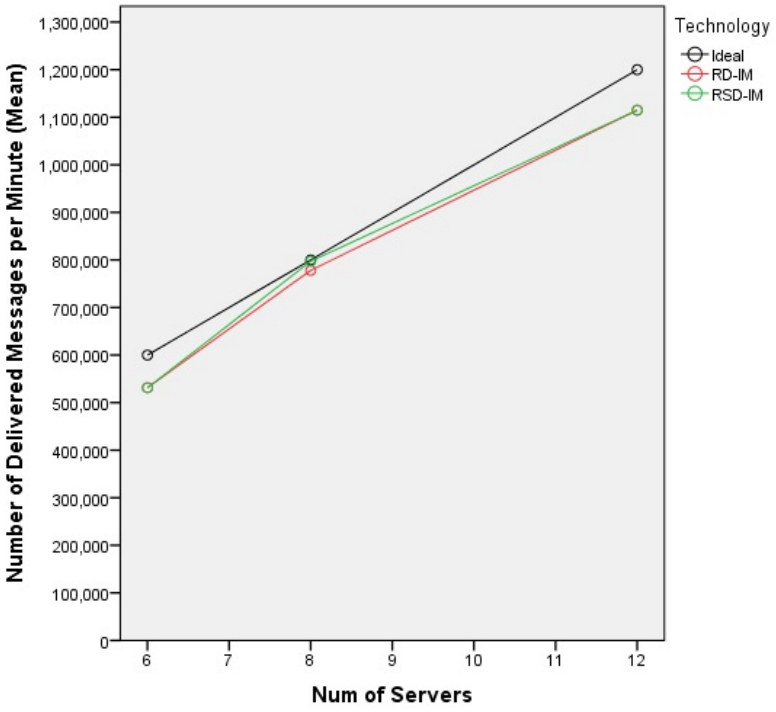

Figure 5.4: D-IM vs SD-IM. Maximum mean throughput when applications are stable.

This is not the only effect observed. Figure 5.4 also shows the ideal growth rate as the number of server nodes scale. Figure 5.2 shows that in the case of six and eight servers, the throughput of both D-IM and SD-IM are above this ideal. However, the introduction of one more router node seems to impact the maximum throughput of the applications, according to the throughput rates described by Figure 5.4.

This is clear in the case of both D-IM and SD-IM, where the throughput is below the ideal for all the considered conditions. In this sense, Experiment 1 showed that in the case of architectures with one router and 12 server nodes throughput was below the ideal as well. However, when a second router node is introduced, the observed throughput for both applications is not only below the ideal, but also lower than the case of only one router architectures (cf. Figure 5.2).

5.4 Experiment 3: Impact of the Number of S_groups with No Failures

Design. This third experiment intends to determine the differences in throughput between the D-IM and the SD-IM when the number of routers varies, but the number of servers is kept constant. As in the case of the Experiment 2 this is a 2×2 factorial design with factors *technology* –namely, Distributed Erlang and Scalable Distributed Erlang— and *number of router nodes* –with four levels: 1, 2, 3, and 4.

The number of server nodes was fixed to 12. This allows the creation of diffferent number of *s_groups* in the SD-IM that vary in size according to the number of router nodes present in each of the experimental conditions. This way, the *s_groups* in the case of the SD-IM were composed of one router and 3, 4, 6, or 12 servers, depending on the level of the variable *number of router nodes* under test. As in the case of the previous experiments, reliability is not considered in this experiment.

Results. Figure 5.5 shows the evolution of the throughput, in a 15 minutes long time series, for the different architectures considered. As in the previous cases the throughput becomes stable after 6 minutes.

The results obtained in this experiment seem to confirm the existence of an effect of the number of routers and the number of *s_groups* in the throughput of the IM applications. Figure 5.5 and Figure 5.6 show that the highest throughput correspond to the IMs that only have one router. As the number of routers increase, the throughput decrease. Yet this decline is different attending to the type of technology involved. This way, the difference in the throughput between a D-IM architecture composed of one router and 12 servers, and another composed of two routers and 12 servers is roughly a 6%, and statistically significant ($p < 0.05$). However, the differences between

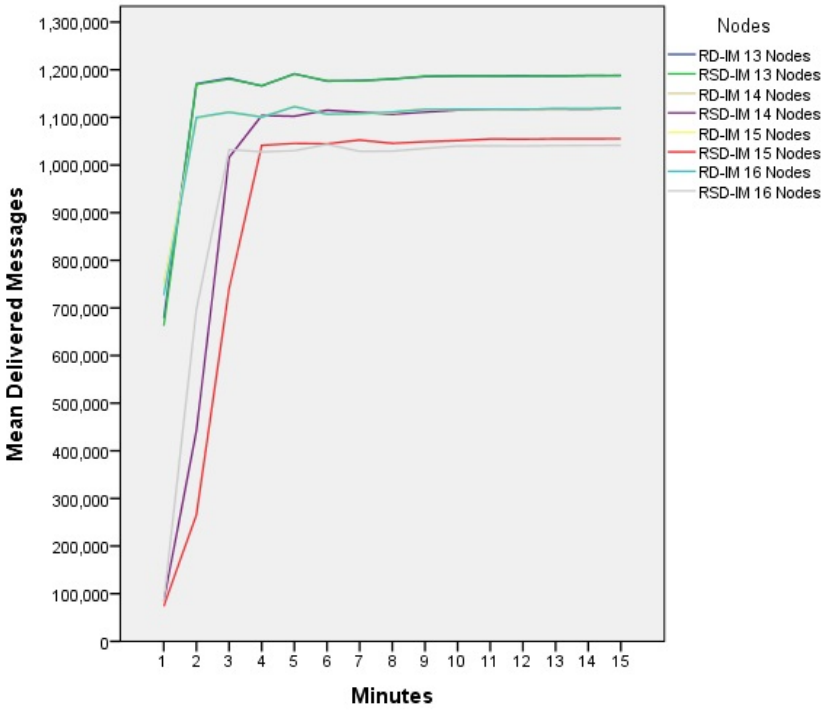

Figure 5.5: D-IM vs SD-IM. Maximum throughput time series.

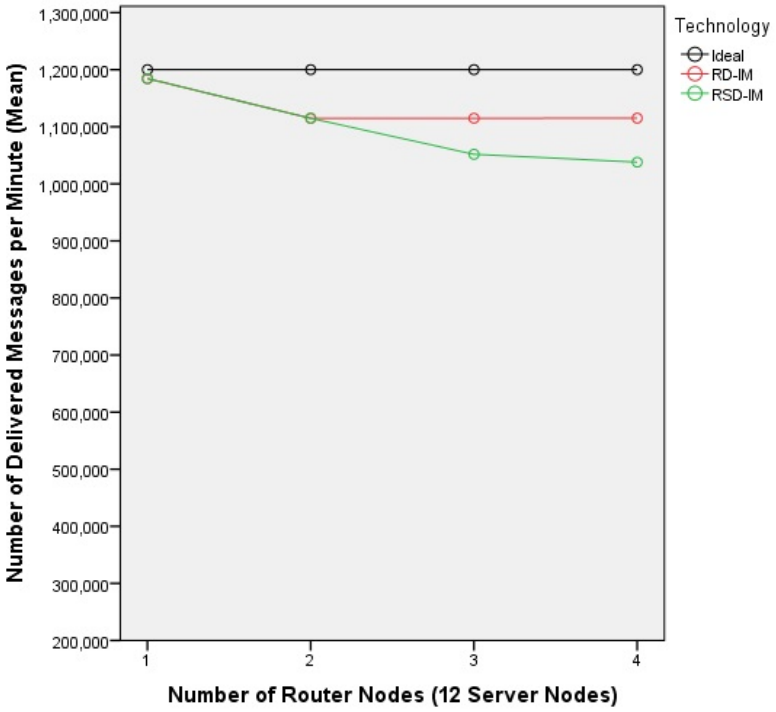

Figure 5.6: D-IM vs SD-IM. Maximum mean throughput when applications are stable.

the two routers architecture and the subsequent three and four routers architectures are minimal and non-significant ($p > 0.05$). In fact, Figure 5.6 show how D-IM throughput remains constant as the number of routers grows.

This *routers effect* is more evident in the case of the SD-IM. Indeed, in this case the addition of a new router node causes a statistically significant reduction of the throughput ($p < 0.05$). In case of the 4 router architecture this difference is around a 12% respect to the 1 router architecture.

Figure 5.6 also suggests that the introduction of new *s_groups* penalise the throughput of the IM application. When a second router node is introduced, the reduction in the throughput of D-IM and SD-IM is similar ($p > 0.05$). However, the more routers are introduced, the greater the differences in their throughputs ($p < 0.05$ for both three and four routers). If there were no effect of the number of *s_groups* these differences should remain constant and non-significant. Moreover, since the differences in throughput between SD-IM three routers and SD-IM four routers (which implies the presence of three and four *s_groups* respectively) are significant as well ($p < 0.05$), seems clear that the number of *s_groups* has an additional impact in the performance of the IM application.

5.5 Experiment 4: Impact of Failures

Design. In the Experiment 4 we analyse fault-tolerance of the IM applications. We compare two conditions: *control*, in which the system runs free of failures for 15 minutes, and *treatment*, in which different random IM processes are terminated at an interval of 15 seconds (i.e., fault rate is 240 failures per hour). Faults were introduced after five minutes of the start of the experimental trials, once the applications are stable. This means that in the case of the experimental conditions, failures were present only from minute 5 to minute 15. Again there are two levels of the *technology* variable, namely, D-Erlang and SD-Erlang.

The architecture used for this experiment consisted of two router nodes and 12 server nodes, making 14 nodes in total. In the case of the RSD-IM (R meaning *reliable*) this generates an architecture with three *s_groups*: one *router s_group* and two *server s_groups*. This architecture is kept constant for the whole experiment.

Again the dependent variable is throughput, measured as the number of delivered messages in one minute. The traffic load is the maximum supported for these architectures, as determined in Experiment 3.

Results. Figure 5.7 shows the evolution of the throughput during the whole experiment, in the presence and absence of failures, for both RD-IM and RSD-IM applications.

The results show that SD-Erlang can be used to build reliable applications in the same fashion as D-Erlang. Moreover, the use of *s_groups* do not impose any special restriction to the default reliability mechanisms with which Erlang/OTP is shipped. Recall that the RSD-IM is just a refactored version of the RD-IM, which introduces the minimum necessary changes required to use the *s_groups* functionality. Apart from this, no further changes were made – especially regarding the supervision trees and fault-tolerance mechanisms.

Thus, two important considerations can be extracted: firstly, *it is possible to build fault-tolerant applications using SD-Erlang in the same way as with D-Erlang.* Secondly, *s_groups neither impose additional costs, nor require additional mechanisms to support fault tolerance.* These assertions are backed by Figures 5.8 - 5.10.

Figure 5.8 shows a summary of the mean throughputs for all the conditions. As it can be seen, they all are practically identical. In fact, two-samples means tests show that there are not statistically significant differences in throughput between RD-IM and RSD-IM ($p > 0.05$; cf. Figure 5.9) as well as between the presence and absence of faults ($p > 0.05$; cf. Figure 5.10) when the fault rate is 240 faults per hour.

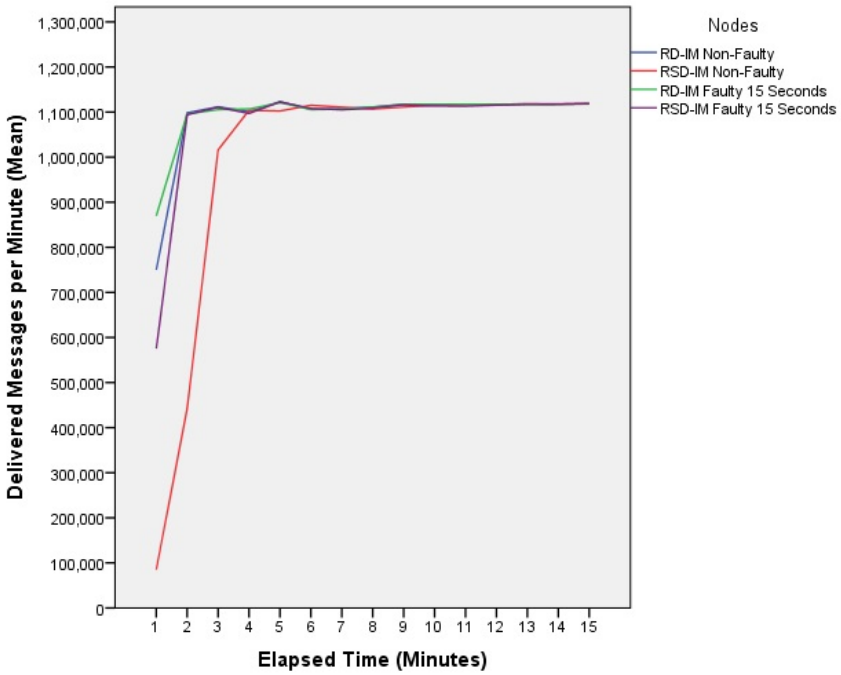

Figure 5.7: RD-IM vs RSD-IM. Throughput time series in absence and presence of faults.

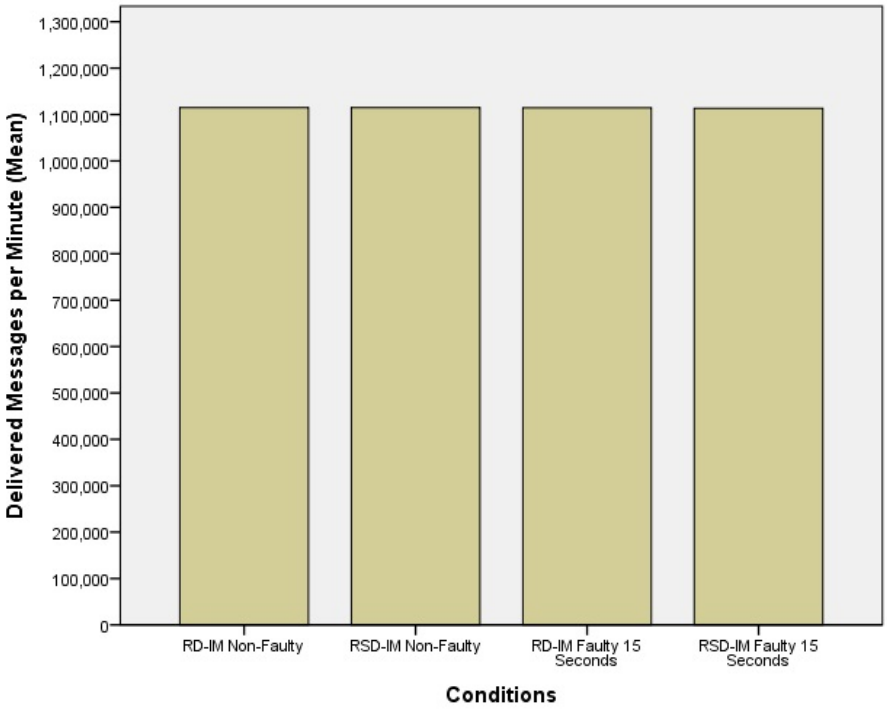

Figure 5.8: RD-IM vs RSD-IM. Mean trhoughput in absence and presence of failures.

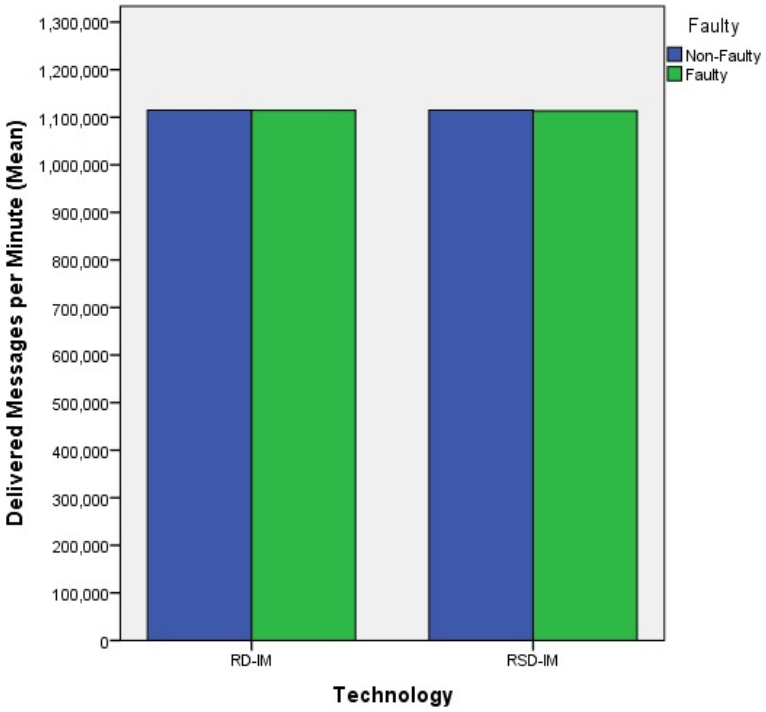

Figure 5.9: RD-IM vs RSD-IM. Mean throughput in absence and presence of faults (intra-technology).

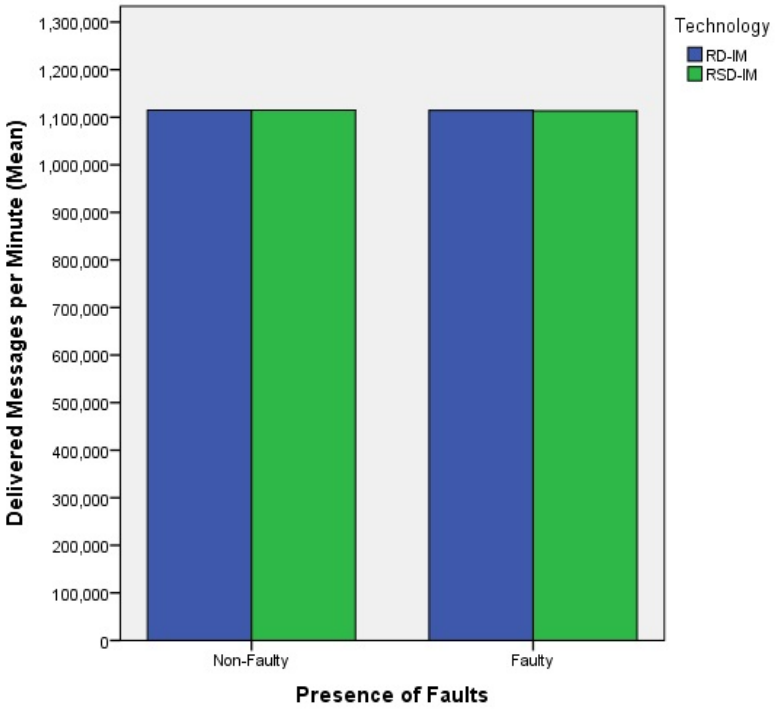

Figure 5.10: RD-IM vs RSD-IM. Mean throughput in absence and presence of faults (inter-technology).

5.6 Experiment 5: Impact of the Rate of Failures

Design. Experiment 5 explores the maximum fault rate supported by the benchmarks. This time all the experimental series introduce faults, but these are introduced, depending on the experimental condition, every 15, 10, and five seconds (i.e., fault rates 240, 360, and 720 faults per hour, respectively). These faults consisted in the termination of random IM processes (one at a time) as in Experiment 4. Two more ad-hoc experimental series were run. In this case, faults consisted of the termination of globally registered databases (i.e., client database and chat session database processes). The faults were introduced at time intervals of five seconds and one second, (i.e., this means that fault rates were 720 and 3600 faults per hour, respectively).

The reason for focusing the faults on the globally registered processes is justified based on one of the SD-Erlang *raisons d'ere*: to improve scalability substituting global operations for *more local* analogous *s_group* operations.

In this case the dependent variable is again throughput measured as delivered messages per minute. The independent variables are *technology* (two levels: D-Erlang and SD-Erlang) and *fault rate* (three levels: 240, 360, and 720 faults per hour). As in the case of Experiment 4, faults were introduced after five minutes of the start of the experimental series, once the applications are stable. For the *ad-hoc* experimental series fault rate has two levels only: 720 and 3600 faults per hour.

Systems architectures and systems loads are the same as in Experiment 4.

Results. Figure 5.11 shows the evolution of throughput at the different fault rates considered for both RD-IM and RSD-IM.

These results together with the results plotted in Figure 5.12, indicate that RD-IM and RSD-IM performances are similar for all the fault rates considered. The exception may be perhaps 3600 faults per hour when only registered databases are terminated (1 Seconds DB only) where RSD-IM appears to have a slightly smaller throughput than RD-IM.

In fact, a two independent samples analysis on *technology* variable only shows that there are not statistically significant differences between them in throughput ($p > 0.01$; cf. Figure 5.13. This analysis considers the throughput means for all conditions grouped by the two levels of the variable *techology*).

A similar test on the variable *type of failure* shows that, initially, there are not differences in throughput, no matter whether the faulty processes are only name-registered processes or not ($p > 0.01$; cf. Figure 5.13).

A two-way ANOVA with factors *technology* and *fault rate* also confirms that there are not statistical effects of any of them nor their interaction on the throughput ($p > 0.01$). Thus, it is safe to affirm that both RD-IM and RSD-IM are fault tolerant, and survive in the same way all the faulty

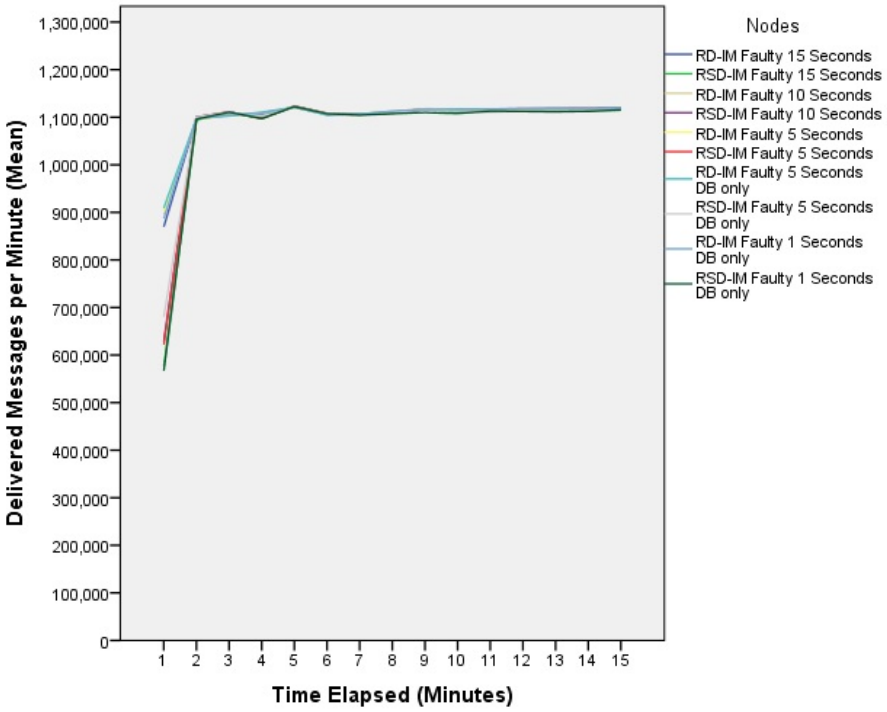

Figure 5.11: RD-IM vs RSD-IM. Throughput time series at different fault rates.

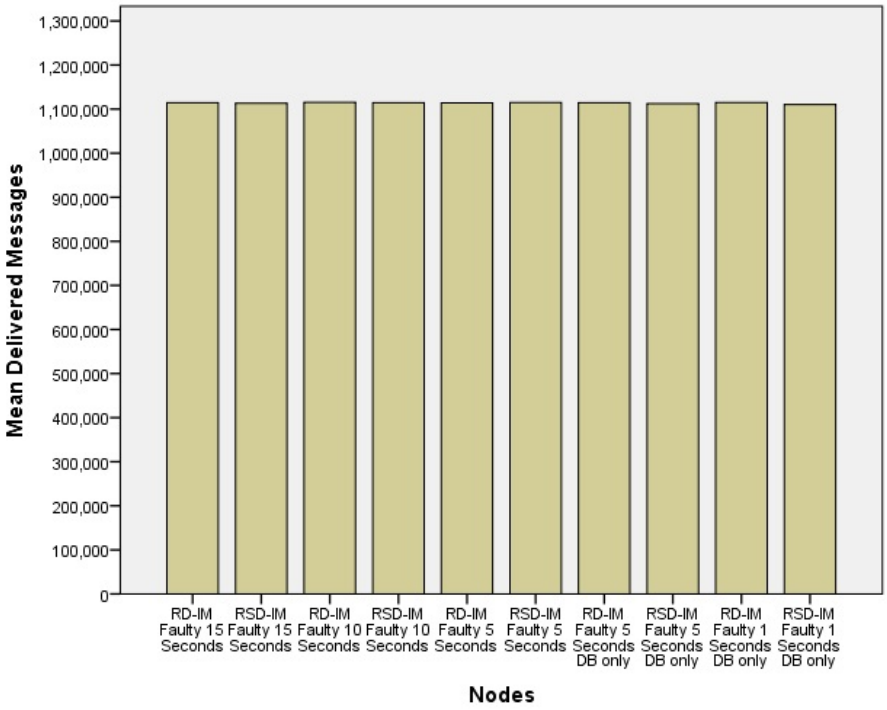

Figure 5.12: RD-IM vs RSD-IM. Mean trhoughput in absence and presence of failures.

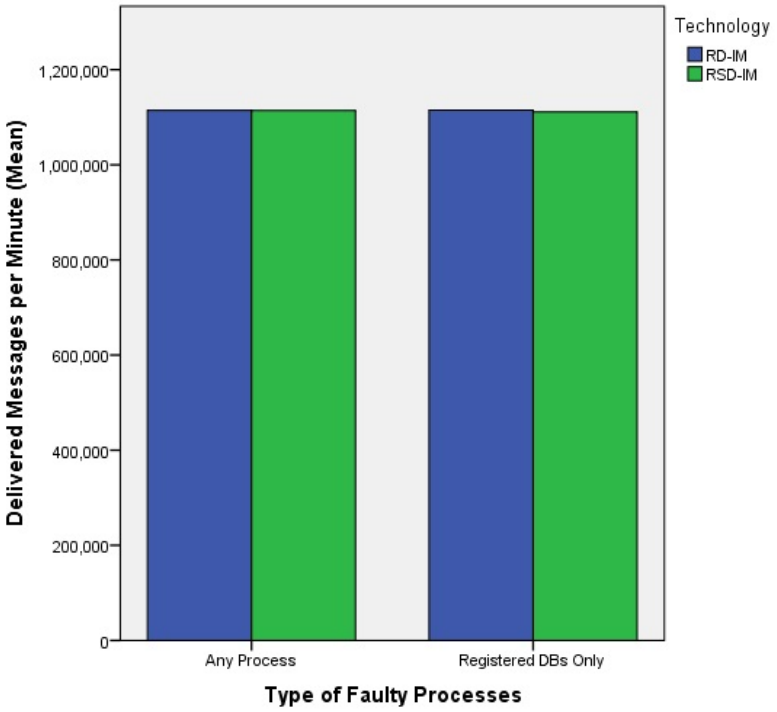

Figure 5.13: RD-IM vs RSD-IM. Mean throughput in when faulty processes are any IM process and registered databases only.

conditions considered. These results together with the results of Experiment 4 also indicate that RD-IM and RSD-IM do not get any penalty in performance when fault rates are as high as 3600 faults per hour.

It was mentioned earlier that RSD-IM seemed to have a slightly smaller throughtput when the fault rate was 3600 faults per hour and the faulty processes were the registered databases only. Figure5.13 shows also a very tiny step when only these faulty processes are considered (columns on the right). This seems to suggest that even though RSD-IM is fault tolerant, its performance in terms of throughput is worse than the RD-IM performance.

To explore this circumstance further, two additional Mann-Whitney non-parametric tests were carried out: one to check whether there are differences between RD-IM and RSD-IM when the fault rate is 720 faults per hour, and the other to make the same check when the fault rate is 3600 faults per hour. None of them were statistically significant ($p > 0.01$). Hence, the difference shown in the graph must be due to the size of the sample ($N = 10$ for each of the four conditions).

Chapter 6

Conclusion and Future Work

In this report we present design and evaluation of two versions of an IM application. The first version (D-IM) is implemented in distributed Erlang. The second version (SD-IM) is a result of refactoring of D-IM with minimal required changes. While the purpose of the IM application is two benchmark distributed Erlang and SD Erlang, the design complies with all main principles of an IM application. The traffic generator was also built to closely mimic human interaction in an IM application (Section 2).

This is also the first benchmark where we evaluate impact of failures in distributed and SD Erlang applications. The results show that on a small scale (up to 12 nodes) RD-IM and SD-IM behave identically, and SD Erlang applications require no additional reliability mechanisms. Thus, the benchmarks behave identically event when killing one globally registered database per second over a 15 minute period (Figure 5.11).

Bibliography

[Arm07a] Joe Armstrong. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[Arm07b] Joe Armstrong. What's all this fuss about Erlang?, 2007.

[AVW92] J. L. Armstrong, S.R. Virding, and M. C. Williams. Use of prolog for developing a new programming language. In *Proceedings of the First Conference on the Practical Application of Prolog*. Association for Logic Programming, 1992.

[CLG+15] Natalia Chechina, Huiqing Li, Amir Ghaffari, Simon Thompson, and Phil Trinder. Improving network scalability of Erlang. 2015.

[CLTG14] N. Chechina, H. Li, P. Trinder, and A. Ghaffari. Scalable SD Erlang computation model. Technical Report TR-2014-003, The University of Glasgow, December 2014.

[CT09] Francesco Cesarini and Simon Thompson. *ERLANG Programming*. O'Reilly Media, Inc., 1st edition, 2009.

[CVss] Francesco Cesarini and Steve Vinoski. Designing for scalability with Erlang/OTP. implementing robust, fault-tolerant systems. In press.

[XGT07] Zhen Xiao, Lei Guo, and John Tracey. Understanding instant messaging traffic characteristics. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*, pages 51–51. IEEE, 2007.